Location-Based Tracking of Moving Objects with Apache Spark

Siva Ravada
Senior Director of Engineering
Spatial and Graph
Solution Overview
Which Vehicle Should Respond to Alarm?

A: 12.3 miles
B: 24.5 miles
C: 11.8 miles
D: 16.2 miles
Solution Architecture

Hadoop Ecosystem

HDFS

Stations Info <GeoJSON>

Nearest Trucks for Equipments <GeoJSON>

Spark Streaming

Streaming Job

Truck Spatial Spatial DStream

Vehicles Location Service

read

stream

in

out

Visualization
Apache Spark

• Apache Spark is a lightning-fast cluster computing technology, designed for fast computation

• It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations
  – includes interactive queries and stream processing

• The main feature of Spark is its **in-memory cluster computing** that increases the processing speed of an application

• **Supports multiple languages** – Spark provides built-in APIs in Java, Scala, or Python
  – Therefore, you can write applications in different languages. Spark comes up with 80 high-level operators for interactive querying
Components of Spark

Spark SQL
Spark Streaming
MLib (machine learning)
GraphX (graph)

Apache Spark Core
About Spark Streaming

• Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.
  – It ingests data in mini-batches and performs RDD (Resilient Distributed Datasets) transformations on those mini-batches of data.

• Data can be ingested from multiple sources (Kafka, TCP sockets, etc).

• Input data from streams can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.

• Processed data can be pushed out to filesystems, databases, and live dashboards.
Resilient Distributed Dataset (RDD)

• A distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.

• An RDD has the following properties:
  – Distributed: RDD’s data is distributed across the network in partitions. Each partition contains a subset of the RDD’s data and can be processed independently by a node in the cluster.
  – Resilient: This feature provides the fault-tolerant capability to an RDD. An RDD can be evaluated at any moment thanks to a lineage graph which contain all the transformations needed to get the current RDD’s data. An RDD can be seen a series of chained transformations over one or more data sets.
  – Lazy evaluation: The data generated by an RDD is not calculated until an action is executed. An RDD can be generated from previous transformation over other RDDs but not a single RDD is evaluated until an action is executed.
Interactive Operations in Spark
Iterative Operations in Spark

[Diagram showing iterative operations involving MR1, MR2, and MR3, with data flowing through iterations and processes like HDFS read and write.]
## Transformation and Materialization of RDDs

1. **Words RDD**
   - **Node 1**
     - Dog
     - Cat
     - Dog
   - **Node 2**
     - Cat
     - Horse
     - Cat
   - **Node 3**
     - Dog
     - Mouse
     - Horse

   - **mapToPair Transformation**

2. **Words Pair RDD**
   - **Node 1**
     - Dog: 1
     - Cat: 1
     - Dog: 1
   - **Node 2**
     - Cat: 1
     - Horse: 1
     - Cat: 1
   - **Node 3**
     - Dog: 1
     - Mouse: 1
     - Horse: 1

   - **reduceByKey Action**

3. **Materialized Data: Word Counts**
   - **Node 1**
     - Dog: 3
     - Cat: 3
   - **Node 2**
     - Horse: 2
   - **Node 3**
     - Mouse: 1
About BDSG Vector API for Spark

• BDSG Vector Analysis API features for Apache Spark

• **Spatial RDD**: Provides both, a SpatialRDD and a distributed spatial index which allows executing spatial transformations and searches on RDDs with spatial information.

• **Spatial Spark SQL**: Spatial functions and optimizations are available for performing spatial queries, similar to the ones performed in RDMS.

• **Structured Streaming**: Spatial UDFs available for Spark SQL are also available for structured streaming.

• **Enrichment**: Geometries can be associated to features from different data layers and utilities are provided for performing geocoding.
Spatial RDD

• A spatial-aware RDD which provides spatial transformations and actions
• Available for both Java (SpatialJavaRDD & SpatialJavaPairRDD) and Scala (SpatialRDD)
• Independent of the data format. Spatial information is retrieved by the SparkRecordInfoProvider interface which may be implemented by the user to extract spatial information from their own data formats.
• Can be spatially indexed to improve search speed
Spatial Transformation & Actions

• Spatial Transformations:
  – flatMap & filter: spatially-enabled overloads of the regular RDD transformations which accept a spatial operator to further filter the results.
  – Spatial join: Joins two spatial RDDs based on a spatial relationship. Available when creating a spatial index for one of the spatial RDDs to be joined.

• Spatial Actions:
  – MBR: returns the minimum bounding rectangle of a spatial RDD
  – nearestNeighbors: returns the K spatial RDD’s records which are closer to a given geometry
Discretized Stream (DStream)

- The basic abstraction provided by Spark Streaming
- Represents a continuous stream of data, either the input data stream received from source, or the processed data stream generated by transforming the input stream. That is, a DStream can be transformed to produce another DStream.
- Processes batches of data which arrive constantly in defined time interval
- DStream is represented by a continuous series of RDDs
Spatial DStream

• A **DStream** that provides spatial transformations
• Available for **Java** (SpatialJavaDStream & SpatialJavaPairDStream) and **Scala** (SpatialDStream)
• Provides the same transformation as a regular DStream plus the following spatial transformations:
  • filter
  • flatMap
  • nearestNeighbors
  • enrich
Building the Solution
Input Data: Stations Data Set

This data set is loaded into memory as plain java objects which later are used to search their nearest vehicles.

- **Type:** static
- **Source:** HDFS
- **Size:** small (fits in memory, ~300 records)
- **Format:** GeoJSON
- **Each station contains the following fields:**
  - **Id:** the unique station identifier
  - **Category:** A number indicating the type of service required
  - **Location:** The longitude and latitude where the station is located. It appears as the geometry field in the stations GeoJSON file
- **In code, each station is represented by the class Station**
Stations Data Set Representations

As file:

```json
{
  "type":"FeatureCollection",
  "collectionName":"equipment",
  "src":8307,
  "geodetic":true,
  "bbox":[-122.40849,37.7972, -122.40016, 37.79769],
  "attr_names": ["id", "category"],
  "attr_types": ["string", "string"],
  "features": [
    {"type": "Feature", "id": "32035878", "geometry": {"type": "Point", "coordinates": [-122.40849, 37.7972]}, "properties": {"id": "32035878", "category": "6"}}
  , {"type": "Feature", "id": "89816696", "geometry": {"type": "Point", "coordinates": [-122.40016, 37.79769]}, "properties": {"id": "89816696", "category": "4"}}
  , {"type": "Feature", "id": "32046146", "geometry": {"type": "Point", "coordinates": [-122.40057, 37.79742]}, "properties": {"id": "32046146", "category": "9"}}
  , {"type": "Feature", "id": "17524196", "geometry": {"type": "Point", "coordinates": [-122.40733, 37.7984]}, "properties": {"id": "17524196", "category": "5"}}
  , {"type": "Feature", "id": "19448008", "geometry": {"type": "Point", "coordinates": [-122.40726, 37.79901]}, "properties": {"id": "19448008", "category": "6"}}
  ]
}
```

In memory: note that the Spatial Java API’s JGeometry class is used to represent the location

```java
public class Station implements Serializable {
    public String getId() {
    ...
    }

    public JGeometry getLocation() {
    ...
    }

    public String getCategory() {
    ...
    }
}
```
Loading Stations Data Set

• Load the stations JSON file from HDFS into a list of Station instances
• The stations will be used later to find their nearest vehicles
  – This data is globally available to all nodes in the cluster

```java
List<Station> stations = null;
try {
    Path stationsPath = new Path(stationsFile);
    FileSystem fs = stationsPath.getFileSystem(conf);
    ObjectMapper om = new ObjectMapper();
    // Load the GeoJSON file as a vector api's FeatureCollection instance
    FeatureCollection fc = om.readValue(fs.open(stationsPath), FeatureCollection.class);
    List<Feature> features = fc.getFeatures();
    stations = new ArrayList<>(features.size());
    for (Feature feature : features) {
        stations.add(new Station(feature.getId(), feature.getGeom(), feature.getProperties().get("category").toString()));
    }
} catch (IOException ex) {
    LOG.error("Problem loading stations from file:" + stationsFile, ex);
    throw ex;
}
```
Service Vehicles Data Set

• This data set is read as a stream from the vehicle location service
  – The service is constantly sending updated locations for each vehicle. This data set is used to look for the nearest vehicles for each station from the stations data set

• Type: dynamic

• Source: TCP service

• Size: ~1000 vehicles which location is constantly updated

• Format: lines containing comma-separated values as follows:
  <vehicle_id>,<category>,<longitude>,<latitude>,<time> where category is the type of service provided by the vehicle and time is the timestamp when the location (longitude and latitude) was updated

• In code, the vehicles data set is represented as a BDSG Vector API’s SpatialJavaDStream containing instances of the ServiceVehicle class
Getting Service Vehicles Data

Stream of data

```
458.5, -122.42316, 37.79109, 4545545
413.5, -122.4174, 37.79099, 4545546
956.5, -122.4172, 37.79004, 4545547
132.5, -122.4185, 37.79643, 4545548
```

Fields per line:
- vehicle_id, category, longitude, latitude, update_time

1) Raw DStream
2) Parsed Vehicle DStream
3) Windowed Reduced Vehicle DStream
4) Spatial Vehicle DStream

Spark Streaming

- Parse as ServiceVehicle instances
- Apply a time window & keep last locations
- Spatially-enable DStream

BDSG Vector API
1. Getting Service Vehicles Data: Read Stream

```java
// Read the stream as comma-separated lines
JavaDStream<String> rawVehiclesStream = streamingContext.socketTextStream(vehicleLocServiceHost, vehicleLocServicePort);
```

Each record is a CSV string

The stream source is a TCP socket
2. Getting Service Vehicles Data: Parse Stream

```java
// Parse CSV lines as Vehicle objects
JavaPairRDD<String, ServiceVehicle> fullVehiclesStream = rawVehiclesStream.mapToPair(csvLine -> {
    ServiceVehicle vehicle = new ServiceVehicle(csvLine, 8307);
    return new Tuple2<>(vehicle.getId(), vehicle);
});
```

Get a DStream containing a key-value pair of vehicle-id, vehicle object

Transform a CSV line into a ServiceVehicle object. Pass the SRID of the vehicles locations (8307)

ServiceVehicle Class

```java
public class ServiceVehicle implements Serializable {
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public JGeometry getLocation() {
        return location;
    }
    public void setLocation(JGeometry location) {
        this.location = location;
    }
    public long getTime() {
        return time;
    }
    public void setTime(long time) {
        this.time = time;
    }
    public void setCategory(String category) {
        this.category = category;
    }
    public String getCategory() {
        return category;
    }
```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.
public class ServiceVehicle implements Serializable{
    
    public String getId() {
        return id;
    }
    
    public void setId(String id) {
        this.id = id;
    }
    
    public JGeometry getLocation() {
        return location;
    }
    
    public void setLocation(JGeometry location) {
        this.location = location;
    }
    
    public long getTime() {
        return time;
    }
    
    public void setTime(long time) {
        this.time = time;
    }
    
    public void setCategory(String category) {
        this.category = category;
    }
    
    public String getCategory() {
        return category;
    }
}
3. Getting Service Vehicles Data: Parse Stream - Problem

- Spark Streaming reads a new batch of data every few seconds, however, a batch of data may not contain information for all the vehicles of the data set.
- Additionally, once a batch is processed, data from this batch is dropped and new data from a newer batch is processed.
- We want to calculate the nearest vehicles for every station using the whole vehicles data set with the latest locations at a given time.

What we would have so far:

- **Batch 1**
  - Vehicle 1-T1
  - Vehicle 2-T1
  - Vehicle 3-T1

- Nearest neighbors:
  - v1
  - v2
  - v3 (Nearest neighbor)
  - Station

- **Batch 2**
  - Vehicle 2-T2
  - Vehicle 4-T2
  - Vehicle 5-T2

- Nearest neighbors:
  - v2
  - v5 (Nearest neighbor)
  - Station

What we need:

- **Batch 2**
  - Vehicle 1-T1
  - Vehicle 2-T2
  - Vehicle 3-T1
  - Vehicle 4-T2
  - Vehicle 5-T2

- Nearest neighbors:
  - v2
  - v1
  - v5
  - v3 (Nearest neighbor)
  - Station
Slicing the input stream into RDDs

DStream

 RDD @ time 1
  data from
time 0 to 1

 RDD @ time 2
  data from
time 1 to 2

 RDD @ time 3
  data from
time 2 to 3

 RDD @ time 4
  data from
time 3 to 4

Sliding window
4. Getting Service Vehicles Data: Windowing & Reducing

Windowing & Reduce

Batch 1
- Vehicle 1-T1
- Vehicle 2-T1
- Vehicle 3-T1

Batch 2
- Vehicle 2-T2
- Vehicle 4-T2
- Vehicle 5-T2

Window: 60 seconds

Batches arrive every 10 seconds

Reduced
- Vehicle 1-T1
- Vehicle 2-T1
- Vehicle 2-T2
- Vehicle 3-T2
- Vehicle 4-T2
- Vehicle 5-T2

Nearest neighbors:
- v1
- v2
- v3
- v4
- v5

Nearest neighbor station

Batch 1 and Batch 2 are windowed and reduced to their nearest neighbors.

Batches arrive every 10 seconds.
Getting Service Vehicles Data: Windowing & Reducing Code

```java
// Create a window to keep all the vehicles in memory. From that window select only records containing the last location (max time value)
JavaPairRDD<String, ServiceVehicle> windowedVehiclesStream = fullVehiclesStream
    .reduceByKeyAndWindow(
        // Keep the vehicle with the max time
        (vehicle1, vehicle2) -> (return vehicle1.getTime() > vehicle2.getTime() ? vehicle1 : vehicle2;)
        // Keep the records from the last windowSeconds
        , Seconds.apply(windowSeconds)
    // Slide the window by batchSeconds
    , Seconds.apply(batchSeconds));
```

This lambda will keep only the last update for a given vehicle:

<table>
<thead>
<tr>
<th>Vehicle 2-T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vehicle 2-T2</td>
</tr>
<tr>
<td>Vehicle 2-T3</td>
</tr>
</tbody>
</table>

| Vehicle 2-T3 |
Getting Service Vehicles Data: 4) Spatially Enabling DStream

The vehicles DStream is transformed into a Spatial DStream so the nearest neighbors transformation can be performed. A Spatial DStream decorates a DStream with spatial features. The only requirement is to provide a way to the Spatial DStream to know the spatial information from the source DStream.

Exposed Spatial Transformations:
- Spatial filter
- Spatial flatMap
- Nearest neighbors
Getting Service Vehicles Data: 4) Spatially Enabling a DStream - Code

```java
//4- create a spatial DStream version from the windowed vehicles stream
SpatialJavaDSpatialStream<
  ServiceVehicle>
spatialVehiclesStream =
  SpatialJavaDSpatialStream.fromJavaDSpatialStream(
    //get only the vehicle object value
    windowedVehiclesStream.map(tup->
      return tup._2();))
  //Specify the SparkRecordInfoProvider to extract spatial information from a Vehicle instance
  ,new ServiceVehicleRecordInfoProvider(srid)
  //We want our SpatialDStream to be of type ServiceVehicles
  ,ServiceVehicle.class);
```

The source DStream is a PairDStream but we are only interested in the value part so a map transformation is applied.

A custom implementation of SparkRecordInfoProvider interface which extracts information from ServiceVehicles instances.
SparkRecordInfoProvider Implementation

```java
public class ServiceVehicleRecordInfoProvider implements SparkRecordInfoProvider<ServiceVehicle> {

    /*
     *
     * private static final long serialVersionUID = 1L;
    private int srid = 0;

    public ServiceVehicleRecordInfoProvider(int srid){
        this.srid = srid;
    }

    @Override
    public boolean getRecordInfo(ServiceVehicle value, SparkRecordInfo recordInfo) {
        recordInfo.setGeometry(value.getLocation());
        return true;
    }

    @Override
    public void setSrid(int srid) {
        this.srid = srid;
    }

    @Override
    public int getSrid() {
        return srid;
    }
}
```

Works with ServiceVehicle instances

Simply get the location from a vehicle.
No more information is required for this example.

Specifies the current SRID of the geometries that are extracted
Getting Nearest Vehicles for Stations
Calculating Nearest Vehicles for Each Station

**Stations**

<table>
<thead>
<tr>
<th>Id</th>
<th>Location</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-122.40849,37.7972</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>-122.40816,37.79769</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>-122.40857,37.79742</td>
<td>9</td>
</tr>
</tbody>
</table>

**Vehicles Spatial DStream**

- Vehicle 1
- Vehicle 2
- Vehicle 3
- Vehicle 4
- Vehicle 5
- Vehicle N

---

Station=1, \{vehicle1, vehicle4, vehicle6\}

Station=2, \{vehicle2, vehicle5, vehicle1\}

Station=3, \{vehicle5, vehicle3, vehicle6\}
Calculating Nearest Vehicles for Each Station: 1) Perform the Nearest Neighbors Transformation

For each station apply the following transformation

```scala
// Spatial operation configuration containing the station location
SpatialOperationConfig spatialOpConf = new SpatialOperationConfig();
spatialOpConf.setQueryWindow(station.getLocation());
spatialOpConf.setTolerance(0.05);

// Get the k nearest neighbors for one station
JavaDStream<List<Tuple2<Double, ServiceVehicle>>> nearestVehiclesStream = spatialVehiclesStream.nearNeighbors(
    truck->{
        int stationCategory = Integer.valueOf(station.getCategory());
        int vehicleCategory = Integer.valueOf(truck.getCategory());
        return vehicleCategory >= stationCategory;
    },
    nearestVehiclesNumber, spatialOpConf);
```

- Specify the geometry the nearest neighbors are looked for.
- Non-spatial filter lambda. Only vehicles with a greater or equal category service number can be considered as nearest neighbors.

Result is a single-value DStream containing a list with a pair of ServiceVehicles and its distances to the current station.
Calculating Nearest Vehicles for Each Station: Transform the nearest neighbors result

The following line transforms the resulting nearest-neighbor stream from a list of distance-vehicle pairs to a key-value pair where the key is the station id and the value is a geoJSON-like containing the information of the nearest vehicles.

```java
//Transform the nearest vehicles result to a pair of the form: station_id, nearest_vehicles_json
JavaPairDS<BigDecimal, String> stationNearestVehiclesStream = nearestVehiclesStream.mapToPair(vehicles -> {
    StringBuilder json = new StringBuilder("{\"features\":[
    int position=0;
    for (Tuple2<BigDecimal, ServiceVehicle> vehicleWithDistance : vehicles){
        if(position > 0){
            json.append(",\n\n";
        }
        position++;
        double dist = vehicleWithDistance._1();
        ServiceVehicle vehicle = vehicleWithDistance._2();
        json.append("{\"type\":\"Feature\", \"id\":\"\"}).append(vehicle.getId())
            .append("\", \"geometry\":\").append(GeoJsonCon.asGeometry(vehicle.getolocation()))
            .append("\", \"properties\":{\"id\":\"\"}).append(vehicle.getId())
            .append("\", \"Position\":\"\").append(position)
            .append("\", \"Category\":\"\").append(vehicle.getCategory())
            .append("\", \"Distance\":\"\").append(formatter.format(dist)).append("\"");
    }
    json.append("\n\n\n"};
    return new Tuple2<>(station.getId(), json.toString());
});

Generate a geoJSON feature for each nearest vehicle

Current station
```
Streaming Job Output

HDFS

Out Folder

Station 1 Folder

Time1 Nearest Vehicles JSON

Station 2 Folder

Time2 Nearest Vehicles JSON

Station N Folder

TimeN Nearest Vehicles JSON

```json
{
  "features": [
    {
      "type": "Feature",
      "id": "1071",
      "geometry": {
        "type": "Point",
        "coordinates": [-122.4235, 37.7912]
      },
      "properties": {
        "id": "1071",
        "Position": "F1",
        "Category": "F1",
        "Distance": "149.01"
      }
    },
    {
      "type": "Feature",
      "id": "2089",
      "geometry": {
        "type": "Point",
        "coordinates": [-122.4266, 37.7910]
      },
      "properties": {
        "id": "2089",
        "Position": "F2",
        "Category": "F1",
        "Distance": "295.70"
      }
    },
    {
      "type": "Feature",
      "id": "3079",
      "geometry": {
        "type": "Point",
        "coordinates": [-122.4174, 37.7909]
      },
      "properties": {
        "id": "3079",
        "Position": "F3",
        "Category": "F1",
        "Distance": "313.59"
      }
    },
    {
      "type": "Feature",
      "id": "4097",
      "geometry": {
        "type": "Point",
        "coordinates": [-122.4173, 37.7909]
      },
      "properties": {
        "id": "4097",
        "Position": "F4",
        "Category": "F1",
        "Distance": "347.76"
      }
    },
    {
      "type": "Feature",
      "id": "5012",
      "geometry": {
        "type": "Point",
        "coordinates": [-122.4155, 37.7964]
      },
      "properties": {
        "id": "5012",
        "Position": "F5",
        "Category": "F1",
        "Distance": "1570.23"
      }
    }
  ]
}
```
Putting Everything Together

- Read (every 10 seconds)
- Parse
- Window (60 seconds) & Reduce
- Spatially-Enable

For each station from GeoJSON file:

Stations
GeoJSON file

Station 1 Nearest Neighbors
Station 2 Nearest Neighbors
Station 3 Nearest Neighbors
Station N Nearest Neighbors

Station 1
NN T1
Station1 NN T2
Station1 NN T3

Station 2
NN T1
Station2 NN T2
Station2 NN T3

Station 3
NN T1
Station3 NN T2
Station3 NN T3

Station N
NN T1
Station4 NN T2
Station4 NN T3
Visualization
Visualization

• The visualization is done using a Web Application which uses the JavaScript Map API.

• Provides the possibility to view the current five nearest vehicles that are able to provide service (based on their category number) to a selected station.

• The user can find the latest records or monitor the records every 10 seconds.
Displaying Stations

Clusters of stations

stations

Cluster zoom in

```
var pointLayer = new OMLayer.VectorLayer("Stations", {  
  def : {  
    type : OMLayer.VectorLayer.TYPE_DATAPACK,  
    url : 'rest/json/getStations',  
    jsonp : true  
  },  
  boundingTheme : true  
});
```
Displaying Nearest Vehicles

Nearest vehicles are displayed for a station. When monitoring is selected, results are updated every 10 seconds.

Latest generated nearest vehicles file is retrieved from HDFS with the help of a simple REST service which is consumed by the Oracle Maps API.

```javascript
vehiclesLayer = new QM.layer.VectorLayer("Vehicles", {
  def : {
    type : QM.layer.VectorLayer.TYPE_DATAPACK,
    url : 'rest/json/findVehicles?stationId='
      + selectedStation,
    jsonp : true
  },
  boundingTheme : false
});
```
Visualization Flow

1 - Display stations in the map

2 – Find or monitor nearest vehicles for station

3 – Display the latest nearest vehicles for the station

HDFS

Stations GeoJSON

Out Dir

Station 1

Time1 Nearest Vehicles JSON

Time2 Nearest Vehicles JSON

TimeN Nearest Vehicles JSON

Station N